The CSx600 micro-computer

The CSx600 is a fictional computer that will be used for examples in class. The architecture of the system is shown in Figure 1, below.

![Figure 1 – CSx600 System Architecture](image)

It has 64K bytes of memory, with an address width of 16 bits, and 10 16-bit registers plus two condition flags. Like most modern computers, memory may be accessed as individual bytes or in multi-byte words, as shown in Figure 2; bytes within a word are stored in little-endian fashion as done by Intel processors:

![Figure 2 – Byte and 16 bit word addressing of memory](image)

Instructions are either a single 16-bit word (2 bytes) for simple instructions, or 4 bytes for instructions which require an additional 16-bit value. Instructions are grouped into the following 9 categories:
● Load, store – move data between registers and memory
● Add, subtract – perform basic arithmetic
● Push, pop – manipulate the stack
● Call, return – subroutine invocation
● Jump – goto another address, either unconditionally or conditionally
● Interrupt – “vectored” subroutine call

2. Detailed instruction definitions

Load/Store instructions:
These operate on 16-bit words and 8-bit bytes, and have the following addressing modes:
● absolute – the address used is given as a parameter to the instruction
● indirect – the address is contained in another register
● indexed – the address is calculated by adding a constant value to another register
● immediate – no address is used, and the value is supplied as part of the instruction.

**LDW_ABS R_{dst}, *addr – load word absolute**
Length: 2 words
Encoding:

<table>
<thead>
<tr>
<th>Opcode = LDW_ABS</th>
<th>R_{dst}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td></td>
</tr>
</tbody>
</table>

Retrieves 16 bits starting at \textit{addr} and puts the value into \textit{R}_{dst}.

**LDB_ABS R_{dst}, *addr – load byte absolute**
Length: 2 words
Encoding:

<table>
<thead>
<tr>
<th>Opcode = LDB_ABS</th>
<th>R_{dst}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td></td>
</tr>
</tbody>
</table>

Retrieves 8 bits starting at \textit{addr} and puts the value into \textit{R}_{dst}. The top 8 bits of \textit{R}_{dst} are set to 0.
STW_ABS $R_{src}, *addr$ – store word absolute
STB_ABS $R_{src}, *addr$ – store byte absolute

Length: 2 words
Encoding:

| Opcode = ST{W|B}_ABS | $R_{src}$ | $addr$ |
|-----------------------|---------|-------|

STW: Takes 16 bit value from $R_{src}$ and stores it into memory starting at $addr$.
STB: Stores 8 bit value found in the low 8 bits of $R_{src}$ into memory at $addr$.

LDW_IND $R_{dst}, *(R_{addr})$ – load word indirect
LDB_IND $R_{dst}, *(R_{addr})$ – load byte indirect

Length: 1 word
Encoding:

| Opcode = LD{W|B}_IND | $R_{dst}$ | $R_{addr}$ |
|-----------------------|---------|---------|

LDW_IND: Fetches a 16-bit word from memory, starting at the address found in register $R_{addr}$ and stores it in $R_{dst}$.
LDB_IND: Fetches an 8-bit byte from the memory address found in register $R_{addr}$ and stores it in $R_{dst}$. The top 8 bits of $R_{dst}$ are set to zero.

STW_IND $R_{src}, *(R_{addr})$ – store word indirect
STB_IND $R_{src}, *(R_{addr})$ – store byte indirect

Length: 1 word
Encoding:

| Opcode = ST{W|B}_IND | $R_{src}$ | $R_{addr}$ |
|-----------------------|---------|---------|

Takes a 16-bit word (8-bit byte) from $R_{src}$ and stores it into memory starting at the address found in $R_{addr}$.

LDW_IDX $R_{dst}, *(R_{addr} + offset)$ – load word indexed
LDB_IDX $R_{dst}, *(R_{addr} + offset)$ – load byte indexed

Length: 2 words
Encoding:

| Opcode = LD{W|B}_IDX | $R_{dst}$ | $R_{addr}$ |
|-----------------------|---------|---------|
|                       |         | $offset$ |

Loads a word (byte) into $R_{dst}$ from the address found by adding $offset$ to the value in $R_{addr}$. 
STW IDX \( R_{src} \), \(*(R_{addr} + \text{offset})\) – store word indexed
STB IDX \( R_{src} \), \(*(R_{addr} + \text{offset})\) – store byte indexed

Length: 2 words
Encoding:

| Opcode = ST{W|B}_IDX | \( R_{src} \) | \( R_{addr} \) |
|-----------------------|-------------|-------------|
| \text{offset}         |             |             |

Stores a word (byte) from \( R_{src} \) into the address found by adding \text{offset} to the value in \( R_{addr} \).

LDW IMM \( R_{dst} \), \text{value} – load immediate value

Length: 2 words
Encoding:

<table>
<thead>
<tr>
<th>Opcode = LDW IMM</th>
<th>( R_{dst} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{value}</td>
<td></td>
</tr>
</tbody>
</table>

Load \text{value} into \( R_{dst} \).
Arithmetic Instructions
These instructions perform arithmetic operations on values in registers. There ought to be byte

**ADD R\_src, R\_dst – add register to register**
Length: 1 word
Encoding:

<table>
<thead>
<tr>
<th>Opcode = ADD</th>
<th>R_src</th>
<th>R_dst</th>
</tr>
</thead>
</table>

Adds the 16-bit value in R\_src to the value in R\_dst and places the result in R\_dst. Z flag is set iff the result is zero; S flag is set iff the sign bit (most significant bit) of the result is 1.

**ADD\_IMM value, R\_dst – add immediate value to register**
Length: 2 words
Encoding:

<table>
<thead>
<tr>
<th>Opcode = ADD_IMM</th>
<th>R_dst</th>
<th>value</th>
</tr>
</thead>
</table>

Adds value to the value in R\_dst and places the result in R\_dst. Sets Z and S flag as above.

**SUB R\_src, R\_dst – subtract register from register**
Length: 1 word
Encoding:

<table>
<thead>
<tr>
<th>Opcode = SUB</th>
<th>R_src</th>
<th>R_dst</th>
</tr>
</thead>
</table>

Subtracts the 16-bit value in R\_src from the value in R\_dst and places the result in R\_dst. Sets Z and S flag as above.

**SUB\_IMM value, R\_dst – subtract immediate value from register**
Length: 2 words
Encoding:

<table>
<thead>
<tr>
<th>Opcode = SUB_IMM</th>
<th>R_dst</th>
<th>value</th>
</tr>
</thead>
</table>

Adds value to the value in R\_dst and places the result in R\_dst. Sets Z and S flag as above.

**MOV R\_src, R\_dst – move (copy) register to register**
Length: 1 word
Encoding:

<table>
<thead>
<tr>
<th>Opcode = MOV</th>
<th>R_src</th>
<th>R_dst</th>
</tr>
</thead>
</table>

Copies the contents of R\_src to R\_dst. Sets Z and S flag as above.
Stack and Subroutine instructions

These instructions are used for manipulating the stack and calling / returning from subroutines.

PUSH $R_{src}$ – push contents of register to stack
Length: 1 word
Encoding:

| Opcode = PUSH | $R_{src}$ |

Subtracts 2 from SP, and then stores the contents of $R_{src}$ to the address in SP.

POP $R_{dst}$ – pop top of stack into register
Length: 1 word
Encoding:

| Opcode = POP | $R_{dst}$ |

Fetches the contents of the memory location indicated by the address in SP, saves it in $R_{dst}$, and adds 2 to SP.

ADD_SP #value – add immediate to stack pointer
SUB_SP #value – subtract immediate from stack pointer
Length: 1 word
Encoding:

| Opcode = {ADD|SUB}_SP | value |

Adds value to SP, thus discarding value/2 elements from the top of the stack. Alternately, subtracts value from SP, reserving value bytes of storage for local variables.

CALL #addr – call subroutine
Length: 2 words
Encoding:

| Opcode = CALL | Addr |

Pushes return address (the address of the next instruction after CALL) onto the stack, and jumps to addr. I.e.: SP = SP-2, *SP = PC+4, PC = addr.

RET – return from subroutine
Length: 1 word
Encoding:

| Opcode = RET |

Pops a return address off the stack and jumps to that address.
Branch instructions
These are unconditional and conditional GOTO instructions, used for e.g. loops and 'if' statements.

JMP #addr – jump unconditionally to address
Length: 2 words
Encoding:

<table>
<thead>
<tr>
<th>Opcode = JMP</th>
<th>Addr</th>
</tr>
</thead>
</table>

Loads the program counter (PC) with \textit{addr}, causing execution to skip to that address.

JMP_Z #addr – jump if zero flag set
JMP_NZ #addr – jump if zero flag clear
Length: 2 words
Encoding:

<table>
<thead>
<tr>
<th>Opcode = JMP_Z / JMP_NZ</th>
<th>Addr</th>
</tr>
</thead>
</table>

If the Z flag is set (not set), jumps to address \textit{addr}, causing execution to skip to that address. Otherwise does nothing.

JMP_NEG #addr – jump if sign flag set (negative)
JMP_POS #addr – jump if sign flag clear
Length: 2 words
Encoding:

<table>
<thead>
<tr>
<th>Opcode = JMP_NEG / JMP_POS</th>
<th>Addr</th>
</tr>
</thead>
</table>

If the S flag is set (not set), jumps to address \textit{addr}, causing execution to skip to that address. Otherwise does nothing.

INT #nnn – software interrupt number nnn
Length: 1 word
Encoding:

| Opcode = INT | nnn |

Reads the value of \textit{interrupt vector nnn}, the 16-bit value at address \textit{2*nnn}, and performs a subroutine call to that address.
3. Calling conventions

[I didn't have time to write this up nicely.]

The CSx600 CPU uses standard calling conventions, with R7 dedicated as the base pointer:

Arguments are promoted to 16-bit values, and pushed onto the stack starting with the last argument; then the CALL instruction is executed.

The function prologue pushes the old base pointer onto the stack, copies the stack pointer into the base pointer, and then subtracts $nnn$ bytes from the stack pointer where $nnn$ is the size in bytes (rounded to a multiple of 2) of the local variables.

The first, second, etc. function arguments may now be addressed as *(bp-4), *(bp-6), …

The local variables in turn may be addressed as *(bp+2), *(bp+4), etc. Note that these expressions do not change, even though the stack pointer moves up and down while calling subroutines.

The function epilogue restores the original stack pointer by (a) copying the base pointer into the stack pointer, and (b) popping the old value of the stack pointer.

The return value is placed in R0 before returning.
4. Memory-mapped I/O

As shown in Figure 1, addresses from 0000 to EFFF (hexadecimal) are used for normal memory, but the 4KB range from F000 to FFFF is devoted to I/O. What this means is that when the CPU reads or writes an address in this range, the operation will be directed to one of several input/output devices: the frame buffer (for display), keyboard controller, disk controller, or serial terminal controller. The memory map for this region may be seen in Figure 2. Note that there are large undefined sections in this map; the result of reading or writing these addresses is not defined, but is unlikely to be good.
Frame buffer (F000 – F77F)

The frame buffer is a contiguous array of 80x24 = 1920 bytes of memory. Each address is mapped to a location on the screen; the byte stored at that address will be displayed in the corresponding screen location. (the VGA screen used by the PC BIOS and e.g. Linux running in console mode works almost identically to this)

Keyboard controller (F800, F801)

When a key is pressed, the key value is stored in the keycode register (F801) and the status register (F800) is set to 1. After software reads the keycode, it writes a 0 to the status register so that it can detect the next keypress.

Serial terminal controllers (F820-F82F)

In order to allow multiple users to access the computer at once, there are four serial ports connected to external terminals. Incoming data from a terminal is received in the same way as for the keyboard controller – the data byte is placed in the data(in) register by the hardware, and status(in) set to 1; the status flag is then set to 0 by software. To send a byte to the terminal, it is written to the data(out) register, and the cmd/status(out) register is set to 1; after the data has been transmitted, the hardware will set the cmd/status(out) register to 0. Note that there are 4 sets of terminal control registers, one for each external terminal.

Disk controller (F810 – F816)

The disk controller reads or writes a single 512-byte disk block at a time. It has a 16-bit register to hold the block number, and an 8-bit command/status register – a command (read = 0x80, write = 0xC0) is written to the register by software, and a status value (0 = failure, 1 = success) is written to the register by hardware when the command is complete. To read a block, software sets the block number register, writes 0x80 to the command register, and waits until the command completes (indicated by the value in the command/status register changing to 01 for success); the data may then be copied out of the buffer. To write a block, data must be copied into the buffer first; then the block number and command registers are written.